Autocoding State Machine in Erlang: A Case Study of Model-Driven Software Development

Yu Guo

Mads Clausen Institute
University of Southern Denmark
Soenderborg, Denmark

Torben Hoffmann Nicholas Gunder

Motorola A/S
Glostrup, Denmark

Abstract

This paper presents an autocoding tool suit, which supports development of state machine in a model-driven fashion, where models are central to all phases of the development process. The tool suit, which is built on the Eclipse platform, provides facilities for the graphical specification of a state machine model. Once the state machine is specified, it is used as input to a code generation engine that generates source code in Erlang.

Keywords: model-driven development, state machine, autocoding, Eclipse, tools

1 Introduction

In Motorola's TWSD organisation, we have been working with Erlang/OTP for a while now and the need for a higher level of abstraction than code has surfaced a couple of times.

The Erlang/OTP code is very clean, but sometimes it is a lot easier to communicate using pictures and models. It is always a practical problem to keep the pictures and models in sync with the code, so any tool support which can help out with that would be appreciated. Even though writing Erlang/OTP code is a lot faster than doing the same code in other languages, you still have to write some boilerplate code to implement a component using Erlang/OTP. As a first step, a state machine model is typically conceptualized in some form before this is done. So a method that could auto-generate code from a state machine model offers some benefits to an Erlang developer.

The semantic gap between Erlang/OTP state machines and formal models of state machines is quite small, and this poses a tough requirement on any modeling tool: To make the tool more useful than writing the code by hand.

The scientific work of Mads Clausen Institute for Product Innovation, University of Southern Denmark was brought to the attention of Motorola, and the potential for solving the problems outlined above seemed so promising that a case study was initiated. [11]

Traditional methods used to develop software are plagued by the problem that the implementation is not always consistent with the specification. Model-driven software development [1][3] seems promising to give the solution, since the implementation can be derived from, or generated directly from the specification. The method requires a modeling language for specifying the application and a code generation engine that translates application models into code. However, it needs adequate tools that automate the steps between specification and implementation.

This paper is intended to give a solution of above problem. When thinking of model-driven software development, the immediate understanding is that models drive the development of the state machine, in the sense that the state machine is constructed by transforming models from higher levels of abstraction to the point where we reach a piece of executable code.

The *autocoding tool suit* is built on the Eclipse platform. Thanks to the wealth of modeling approaches the platform supports, most of which are based on well-established and popular projects. We use the Eclipse Modeling Framework (EMF) project as the modeling facility and the Eclipse Graphical Modeling Framework (GMF) project to provide a graphical modeling environment. The tool supports both a textual notation as well as a visual one. The Acceleo can be fruitfully exploited for a transformation engine to develop the tool for the code generation. It has built-in facilities to read models that support the smooth integration with modeling tools in the Eclipse. Developed in such way, the state machine autocoding tool suit contains a set of Eclipse plug-ins, therefore, a uniform development environment can be obtained.

The rest of the paper is structured as follows: Section 2 presents state machine used in Erlang. Section 3 deals with the metamodel and constraints of the state machine. Section 4 describes the generation template. The implementation in Eclipse is presented in Section 5. Section 6 discusses the development process using the tool. A future work is discussed in Section 7, and a summary is given in the concluding section of the paper.

2 Finite state machine in Erlang

The finite state machine used in Erlang is described as a set of relations of the form:

```
State(S) x Event(E) -> Actions(A), State(S')
```

The relations are interpreted as meaning: If we are in state `S` and the event `E` occurs, we should perform the actions `A` and make a transition to the state `S'`.[4]

The finite state machine can be further considered as a *mealy state machine* [14], where an output (or action) is generated based on its current state, and an input (or event). A transition is marked with a trigging event, a guard expression and an action. The guard is a boolean expression. A developer should think in the following manner: if an event associated to a transition occurs, and the guard on

the transition is satisfied, the transition is fired. Consequently, the state machine reacts to the event by performing the action on the transition; state maybe changed, too.

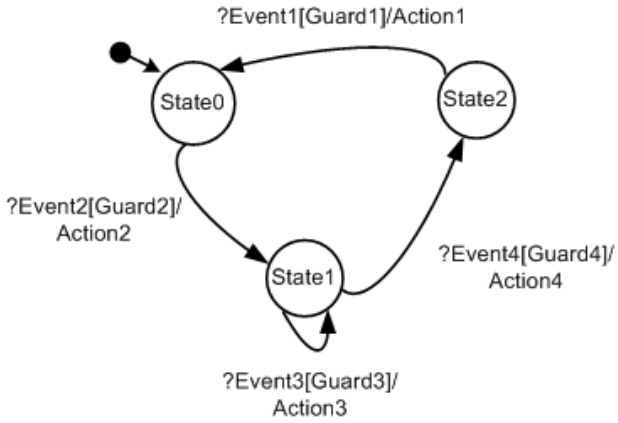

Fig. 1. The mealy state machine model

According this state machine model, for each transition of a state, a transition clause should conform to the following convention, when using the Erlang gen_fsm behaviour to implement the state machine:

```
StateName(Event, StateData) when Guard ->
      ...  code for actions here ...
{next_state, NextStateName, NextStateData}
```

3 Metamodeling the state machine

Metamodeling plays a fundamental role when using the model-driven software development approach. A metamodel describes possible structure of models, by defining the constructs and their relationship of the modeling language, as well as constraints. It is also the basis for building tools, concerning construction of the state machine model, validation of models against constraints, as well as generation of code. A constraint specifies a restriction of the metamodel element it is applied to. It can be written in natural language or in the Object Constraint Language (OCL) [5].

To define a metamodel, a metamodeling language is required. The Eclipse Modeling Framework Project provides facilities to create a metamodel, with the support of a meta-meatmodeling language – *the Ecore metamodel*. The language is considered to be at the M3 layer of the Meta-Object Facility (MOF) Four Layer Metadata Architectures [2]. In EMF, a metamodel described by the Ecore metamodel, known as *Ecore model*, contains structural requirements and constraints for the model, which are information contents that the model editor will manipulate. It extends the Ecore metamodel by instantiating classes, in the sense that a new class with new attributes in the Ecore model is created as an instance of an existing one defined in the Ecore metamodel. A metamodel is at the M2 layer of the MOF Four Layer Metadata Architectures. (see Fig. 2)

The Fig. 3 presents the metamodel of the mealy state machine. The metamodel is fairly intuitive and easy to understand. The state machine container is the root

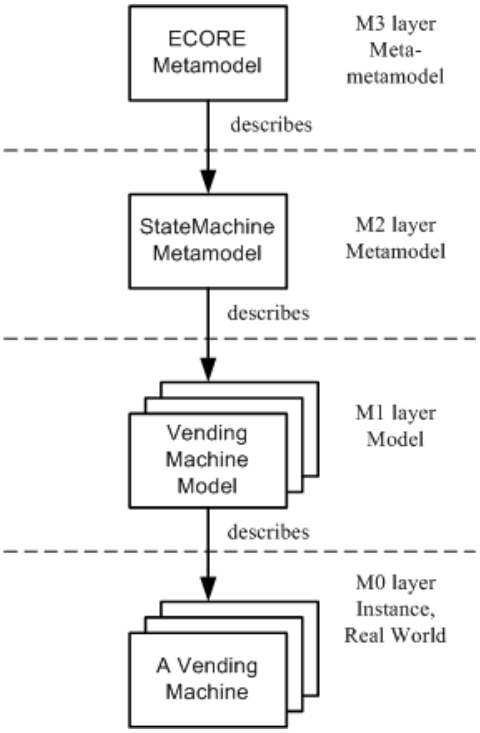

Fig. 2. MOF Four Layer Metadata Architectures

element of all instances. It must contain one state machine instance, and an arbitrary number of action as well as event's instances. The instances of the action and the event are referenced by transitions of the state machine model.

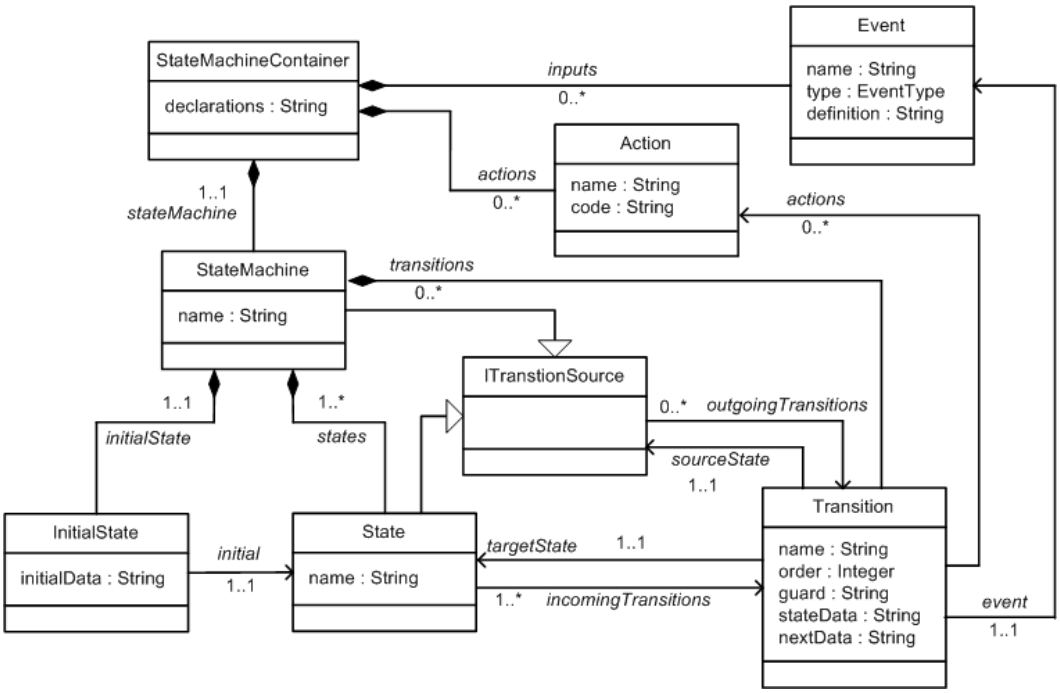

Fig. 3. The state machine metamodel

Typically, a transition has state instance as "sourceState" or "targetState". But, the state machine in Erlang has the feature that an event can arrive at any state. The event is sent with the `gen_fsm:send_all_state_event/2` instead of sending the event with `gen_fsm:send_event/2`, so that only one clause `Module:handle_event/3` is needed to handle the event [4]. In order to model this feature, an event-triggered transition can have the state machine as source, meanwhile, the type of the event on this kind of transitions must be "all_state_event". That is why both the "StateMachine" and the "State" class implement the interface "ITransitionSource".

The state machine in Erlang is deterministic, as the sequence of the clause for each transition matters while executing the program. The clauses are to be matched according to the order of precedence in the source file. If the first match failed, this second one will be picked. This is why an "order" attribute is used on the transition of the state machine model. It is an integer value. All outgoing transitions of a state must have different order. It determines the appearance sequence of the transition clauses in the source file at the generation stage.

To obtain a complete domain model, the metamodel needs to be accompanied with constraints. It will not allow you to perform a code generation without doing the series of static checks. The constraints in a model-driven fashion are categorized into two levels: the platform independent level and the platform specific level. The platform independent constraints have not concerns with which target code is going to be generated, whereas the platform specific ones are bound to the target language, in this case study – the Erlang language. But in case of creating an Erlang state machine specific model, constraints from two levels can be combined.

For instance: according to the metamodel, both "State" and "StateMachine" are subclass of the "ITransitionSource" so that they can have outgoing transitions. However, the state machine turns into dead end if there were not any outgoing transition from a state. Thus a constraint like "each state has at least one outgoing transition" is a platform independent one. On the other hand, "the names of the states and the events must start with lowercase letter" is an Erlang platform specific constraint, since this rule has to be satisfied to make the generated code compliable. In case of C code generation from the state machine model, the constraint is not necessary.

4 Generation Template

After the metamodel has been built, a model, which is instance of the metamodel, can be used together with the templates for generation. Having models as central to all phases of the development, the generation process is independent from the concrete syntax of the model. No matter which format a model is stored as, the metamodel is always of special significance in the context of model-driven development. Generation templates should not be written based on some specific format that the model stored as, but on the metamodel.

To make the autocoding feasible, the way of programming a finite state machine needs to be normalized. The result of the normalization plays the role as the generation template. This step requires the answer of the question: where the

static code, dynamic code and manual code are located in the template. The static code is always the same for all different models, whereas the dynamic code is the one transformed from the model.

```
STATE_NAME (EVENT_NAME1,STATE_DATA1) when GUARD1 ->
            GENERATED ACTION1,
            %%manual code for action
            NextState = TARGETSTATE_NAME1,
            NextStateValue = DATA1 %%or manual code
{next_state, NextState, NextStateValue};

STATE_NAME (EVENT_NAME2,STATE_DATA2) when GUARD2 ->
            GENERATED ACTION2,
            %%manual code for action
            NextState = TARGETSTATE_NAME2,
            NextStateValue = DATA2 %%or manual code
{next_state, NextState, NextStateValue}.
```

Fig. 4. The generation template

The piece of pseudo code (Fig. 4) reveals the structure for one state with two outgoing transitions in the generated code. Words all in uppercase are dynamic code, which will be generated from the model. As mentioned in the previous section, the sequence of the two clauses depends on the value of the order attribute of outgoing transitions. The value of the variable `NextStateValue` could be generated from the "nextData" attribute of the transition if specified, or be written manually if it is empty, which is dependent of whether or not this value should be calculated by manual code. The `STATE_NAME` for two transition sections must be identical because they represent the same state. The rest dynamic code such as `EVENT_NAME`, `STATE_DATA` or `GUARD` could be either the same or different for each outgoing transitions. No manual code is allowed between the assignment of NextStateValue and the statement `next_state, NextState, NextStateValue`. It is only allowed to add manual code to the specified place. Otherwise, it will be lost in case of regeneration.

5 Implementation

The model-driven software development approach has been adopted to develop the autocoding tool suit in the Eclipse platform (www.eclipse.org), so as to reduce the amount of manual work needed to develop tools in a conventional manner. The Eclipse Graphical Modeling Framework project provides means to ease and speed up the development of graphical editors for modeling, which can be used for the rapid development of standardized Eclipse graphical modeling editors, by providing a generative component and runtime infrastructure for developing graphical editors [10].

The majority of the state machine model can be created graphically, but there is something that a graphical editor does not necessarily be applied. The Eclipse provides property sheet as complementary to editors, with basically two functionalities: set or display property of a model; create model instances that cannot (necessarily)

be created using the graphical editors. Therefore models like actions and events can be instantiated within property sheets, otherwise the diagram that is supposed to emphasize the relation between states and transitions will be polluted with those less important ones to the visualization.

The Acceleo (www.acceleo.org) is an Eclipse based code generator transforming models into code. It works with any metamodel, implementing MOF as specified by the OMG (The Object Management Group). It needs templates that describe the information required to generate source code from a metamodel. Acceleo deals with incremental generation, by defining specific protected area. Code, which is modified by developers, is surrounded by special tags in the protected area. These tags do not pollute target code because they are implemented as explicit comments. On the next generation, the whole text in the protected area is kept.

A nice feature that the Acceleo offers is that Java functions can be invoked as services within the generation template. Acceleo is a hybrid with respect to the generation template. The advantages of an expressive template language and the power of the Java language are well combined, in such a way that parts of the template is written in the template language, while the complicated algorithm are implemented in Java.

Fig. 5 shows the graphical development environment in the Eclipse. A vending machine model is built with the editor. In order for a correct code generation process, the model has to be validated. The Fig. 6 presents the case when two states are identically named, thus, with the support of the GMF runtime, the models that violate the constraint are highlighted.

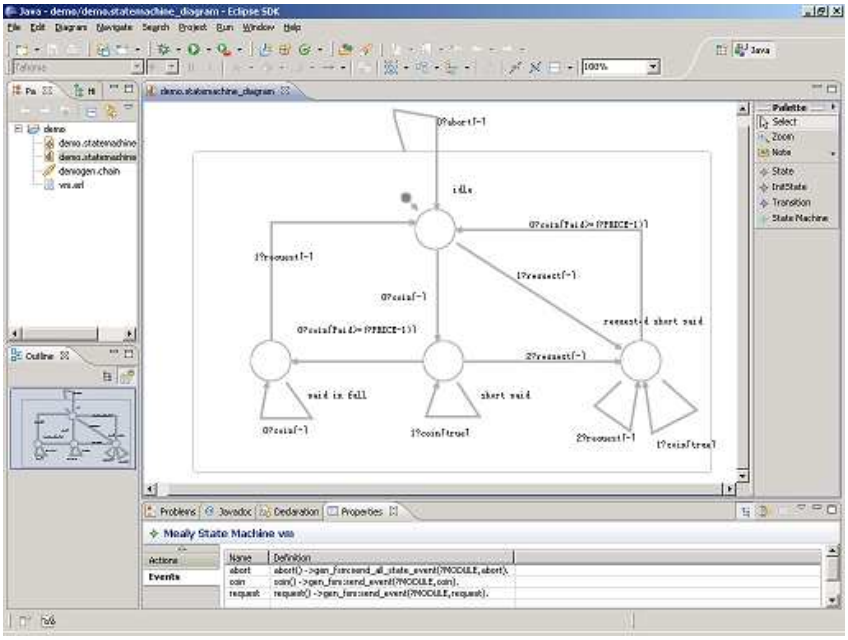

Fig. 5. The graphical development environment

Once the state machine model has been built, the tool can deserialize it into XML format, which gives the ability to maintain interoperability among tools based on the metamodel, as well as the 3rd party tools.

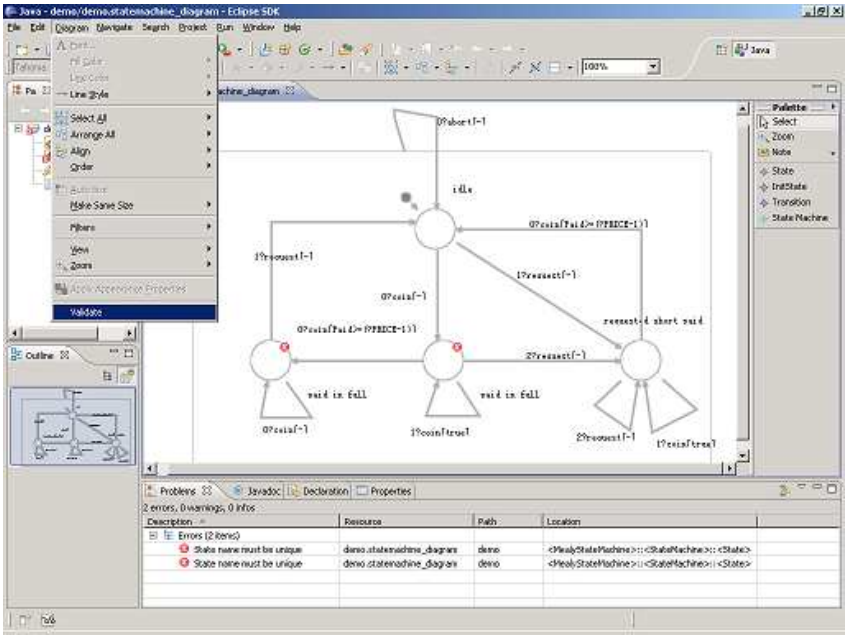

Fig. 6. Violation of a constraint

State machine code will be generated then from the model. The vending machine code and model, as a reference, used for code generation in this case study is not something that Motorola markets, but the vending machine problem and original code is quite similar to most of the state machine code Motorola has written so the results from the case study applies to the real world code as well.

6 Development Process

It is well known that, even when using the model-driven development approach, some code still has to be handwritten. The handwritten code could come from any legacy code that has not been integrated into the model. While the state machine model can express when the actions are invoked, the implementations and algorithms of those actions are not represented. So, after generating the skeletons of the state machine, developer still needs to fill those actions skeletons with business logic.

Usually a balanced approach is considered to be the best, in the sense that generating the initial code base from the model and then just start from there. It can save some of the initial tediousness of translating the model to code. Users should not modify the generated code unless the changes are in the protected area. Otherwise, these changes will be lost in the next generation iteration.

The obvious way to lower the burden of the developer is to generate 100% code from the model. To achieve this goal, an iterative development process can be considered when using the tools. At the beginning, the developer should concentrate on the application logic to design the model. He needs to specify states, transitions, events as well as the names of the action, and put some dummy code (like comments) inside the action method. Once the state machine skeleton has been generated, he needs to fill out the actions method with the business logic code in the protected

area.

When the state machine code passed tests, the handwritten code can be further integrated back into the model by replacing the dummy action code. (This has to be done manually, at the moment.) Consequently these code will be generated in the next generation. Finally, the developer will end up with a state machine model integrated with all action code. In this manner, it is possible to generate complete code if all actions are just predefined function calls from some library.

However, bear in mind that the complete model is not a platform independent model anymore. It becomes a platform independent model mixed with the platform specific code. In case of generation of another target code from this model, for example C code, the platform specific part in the model has to be changed.

It is possible to make improvement by introducing some kind of textual language that specifies the action. In this case the language has to be defined or even be invented. Meanwhile all the tooling needs to be created, too. It is not a trivial task since the semantics of the language has to be correct. As a result, the flexibility is obtained at the cost of the increased complexity.[3]

7 Future Work

The work that has been described in this paper is an initial step towards development tools for application in Erlang/OTP, in a model-driven fashion. It can be continued in several possible directions to further improve developer productivity by introducing more features as described in following sections.

7.1 Configuration Management

One of the biggest practical problems Motorola has experienced with various model driven approaches has been the management of different versions of a model component.

Being large organization with many development centers in the world, Motorola is challenged to share common tools and code with other teams who are not necessarily located in the same region. Typically, these teams have their own additions and changes, sharing a common code base. One problem deals with the task of being able to integrate individually created functionality, as well as that created by separate development centers, into a common code base. The fundamental problem resulting in this becomes an issue of potential rework and many headaches when the same code is changed by different people. Working at the source code level, this is a merging problem which requires a significant about of labor to do consistently. When moving to model-driven approaches, the problem does not go away – it is merely lifted to a higher level of abstraction, where models and changes to models need to be merged. This is a problem that so far has not been solved well enough to be of practical use. The practical problems in this has so far resulted in a situation where models are used to generate the first version of a component, and then all additions and changes are done at the source code level.

Clearly, there are two basic problems that need to be solved in order to make a model-driven approach a good fit for Motorola:

- Integration of manual changes back into the model. (Fig. 7)
- Merging of a new version of a model with the source code based on a previous model plus some manual changes. (Fig. 8)

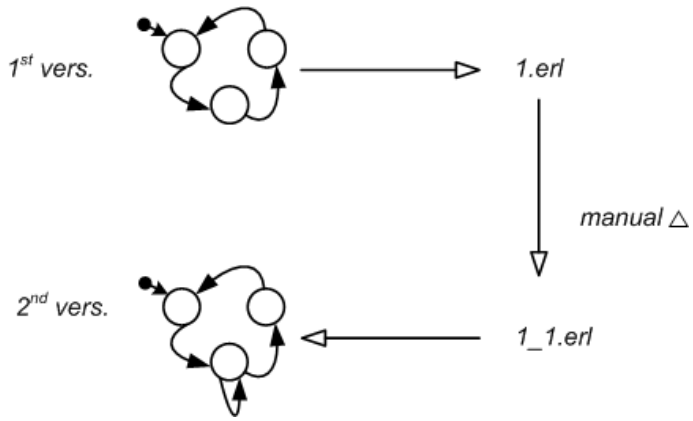

Fig. 7. Integration of manual changes

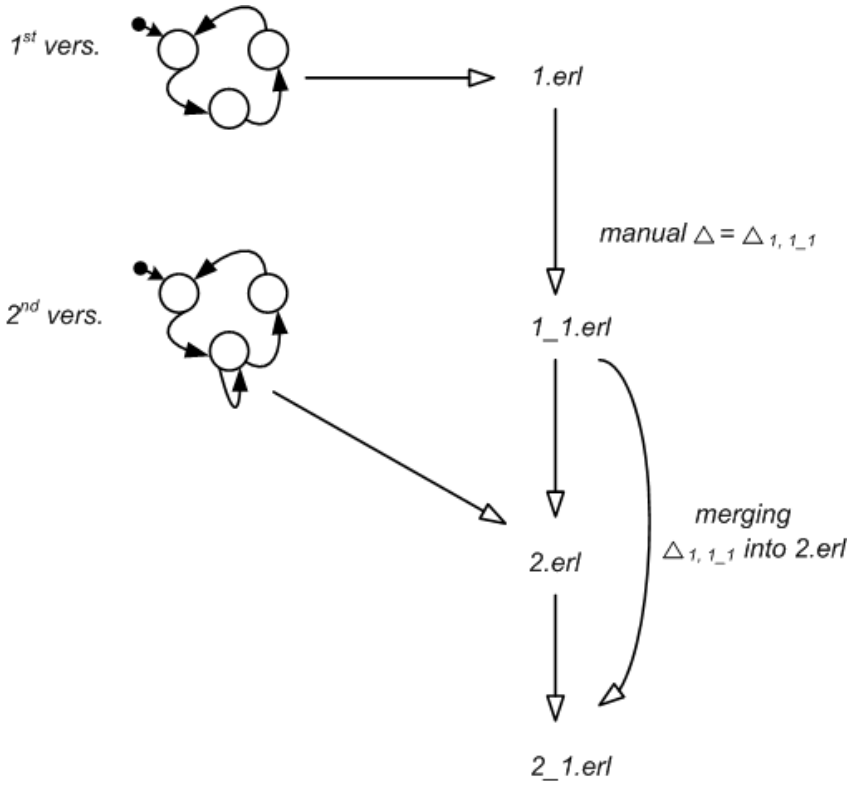

Fig. 8. Merging of a new version of a model with the source code

When these two problems are solved, it becomes possible to solve the general problem of merging two changes to the same model. (Fig. 9)

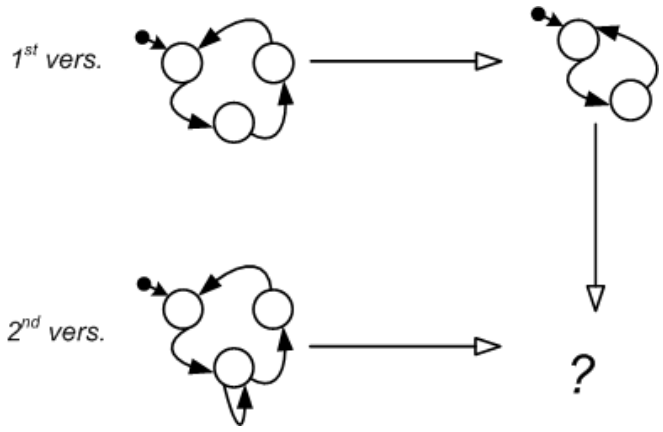

Fig. 9. Merging two changes to the same model

7.2 Refactoring

Sometimes you have an existing code base and would like to introduce a model-driven approach – more often than not, this is a major practical issue since most freely written code does not easily fit with a model framework.

In order to overcome this obstacle it would be interesting to investigate the use of refactoring tools that assist the programmer in transforming a legacy code base to a format that allows easy reverse engineering of the code into a model.

In this case, a given state machine code needs to be transformed back into the model that is at a higher level of abstraction. Adding reverse engineering for the current tool suit makes it possible to offer a visual content of the state machine from legacy code as higher-level document. Applying model based analysis technique i.e., model checking [7] to the model is also feasible, once the model has been built from the legacy code.

Wrangler [8] is a refactoring tool providing a collection of basic refactorings to the program. With the help of the tool, the legacy code can be refactored as close as possible to the generation template. Subsequently, A text-to-model transform engine can be applied to take the refactored program as input and produce the model correspondingly.

7.3 Artifact generation

Besides the state machine source code, more artifacts can be generated from the model such as supervisor, application, makefile, etc., as they require certain information contained in the state machine model, too. This information can be exploited as much as possible in order to minimize the amount of manual work. However, the current state machine model does not necessarily contain all information for these artifacts. Depending on what to generate, proper metamodels, editors and generators could be added to the current development environment. Following the introduced approach, it is not hard to extend the current tool suit with the new components.

If a complete metamodel that covers all the concepts of the domain was developed, equipped with right tools, a component framework and a set of predefined

components, the developer is able to model the whole telecommunication application and generate 100% code from it. The ErlCOM [12], as a domain specific language for robust reconfigurable components, tries to introduce a component layer on the top of Erlang environment, so that the developer concentrate on the system design at a higher level. It provides a useful packaging framework that enables programmers to organize their applications written in Erlang in such a way that they could be easily reconfigured either so that they could adapt in a rapidly changing runtime environment or they could be reused. [13] Modeling and code generation tools have been developed based on the Generic Modeling Environment. However, its modeling concept mainly focuses on one of the static aspects of an application – component configuration management. The dynamic aspect of the application, that could be modeled using state machine, is not yet covered.

7.4 Model debugging

The current tool suit enables the developer to visualize the state machine as a diagram. The diagram and model could be further exploited within a debugging session by providing animation functionality. An executing state machine could send certain information, which is embedded into the generated code, back to the diagram. This information is used to synchronize the graphical notation to trigger the animation. States and transitions with the synchronous data should be highlighted. If the developer recognizes a mistake in the diagram while debugging, he can instantly change the model, and generate the code once again. However, the current stage of the development of the Eclipse modeling projects does not yet provide such a framework for the graphical debugging according to a given metamodel. More research needs to be done in this area.

8 Conclusion

The paper has presented an autocoding tool suit specifically designed to support finite state machine development. It facilitates the development by providing the developer with a specialized graphical editing environment to specify the state machine model, and a code generator to produce Erlang code. An initial prototype of the tools has been developed on the Eclipse platform following the model-driven software development philosophy. The model-driven approach for tool development has been experimented, in which models are central to all phases of the development process.

References

[1] The Object Management Group, *MDA Guide Version 1.0.1*, 12th June 2003

[2] The Object Management Group, *Meta Object Facility(MOF) Specification, Version 1.4*, April 2002

[3] Stahl,T., M. Völter, J. Bettin, A. Haase, S. Helsen, K. Czarnecki (Foreword by), B. von Stockfleth (Translated by), "Model-Driven Software Development: Technology, Engineering, Management", ISBN: 978-0-470-02570-3, Wiley, 2006

[4] Ericsson AB, *OTP Design Principles Version 5.6.4*

[5] Warmer, J, A. Kleppe, "Object Constraint Language, Getting Your Models Ready for MDA", Second Edition, Publisher : Addison Wesley, Pub Date : August 29, 2003, ISBN : 0-321-17936-6

[6] Chikofsky, E.J.; J.H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software*, IEEE Computer Society: 13C17. January 1990

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen, *A Tutorial on Uppaal*, Updated 17th November 2004. Department of Computer Science, Aalborg University

[8] Huiqing Li, Simon Thompson, George Orosz, and Melinda Toth, *Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse*, Proceedings of the Seventh ACM SIGPLAN Erlang Workshop, page 12pp. ACM Press, September 2008.

[9] Frank Budinsky, Dave Steinberg, Ed Merks, Ray Ellersick, Timothy J. Grose, "Eclipse Modeling Framework", Published Aug 11, 2003 by Addison-Wesley Professional

[10] Frederic Plante, *Introducing the GMF Runtime*, Eclipse Corner Article, January 16th, 2006

[11] C. Angelov, Xu Ke, Yu Guo and K. Sierszecki, *Reconfigurable State Machine Components for Embedded Applications*, Proc. of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2008, Parma, Italy, Sept 2008

[12] Gabor Batori, Zoltan Theisz and Domonkos Asztalos, *Robust Reconfigurable Erlang Component System*, In The Proceedings of 11th International Erlang/OTP User Conference, Stockholm, Sweden, November 2005.

[13] Gabor Batori, Zoltan Theisz, and Domonkos Asztalos, *Configuration Aware Distributed System Design in Erlang*, In The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden, November 2006.

[14] Mealy, GH, *A Method to Synthesizing Sequential Circuits*. Bell System Technical J, 1045-1079. 1955